The Translation of Functional Programming Languages

11 The language PuF

We only regard a mini-language PuF ("Pure Functions").

We do not treat, as yet:

- Side effects;
- Data structures.

A Program is an expression e of the form:

$$
\begin{array}{rcl}
e & ::= & b \mid x \mid (\Box_1\ e) \mid (e_1\ \Box_2\ e_2) \\
 & \mid & (\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ \ e_2) \\
 & \mid & (e'\ e_0 \ldots e_{k-1}) \\
 & \mid & (\textbf{fn}\ \ x_0, \ldots, x_{k-1} \Rightarrow e) \\
 & \mid & (\textbf{let}\ \ x_1 = e_1; \ldots; x_n = e_n\ \textbf{in}\ \ e_0) \\
 & \mid & (\textbf{letrec}\ \ x_1 = e_1; \ldots; x_n = e_n\ \textbf{in}\ \ e_0)
\end{array}
$$

An expression is therefore

- a basic value, a variable, the application of an operator, or
- a function-application, a function-abstraction, or
- a **let**-expression, i.e. an expression with locally defined variables, or
- a **letrec**-expression, i.e. an expression with simultaneously defined local variables.

For simplicity, we only allow `int` and `bool` as basic types.

Example:

The following well-known function computes the factorial of a natural number:

$$\begin{array}{lll} \textbf{letrec}\ \text{fac} & = & \textbf{fn}\ x \Rightarrow \textbf{if}\ x \leq 1\ \textbf{then}\ 1 \\ & & \textbf{else}\ x \cdot \text{fac}\ (x-1) \\ \textbf{in}\ \text{fac}\ 7 & & \end{array}$$

As usual, we only use the minimal amount of parentheses.

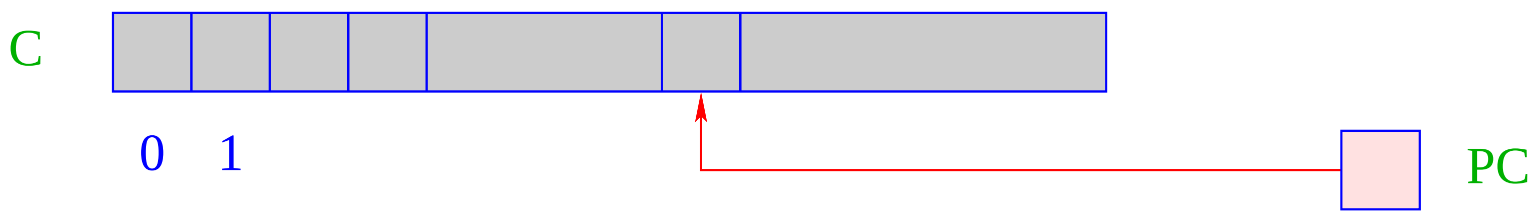
There are two Semantics:

CBV: Arguments are evaluated before they are passed to the function (as in SML);

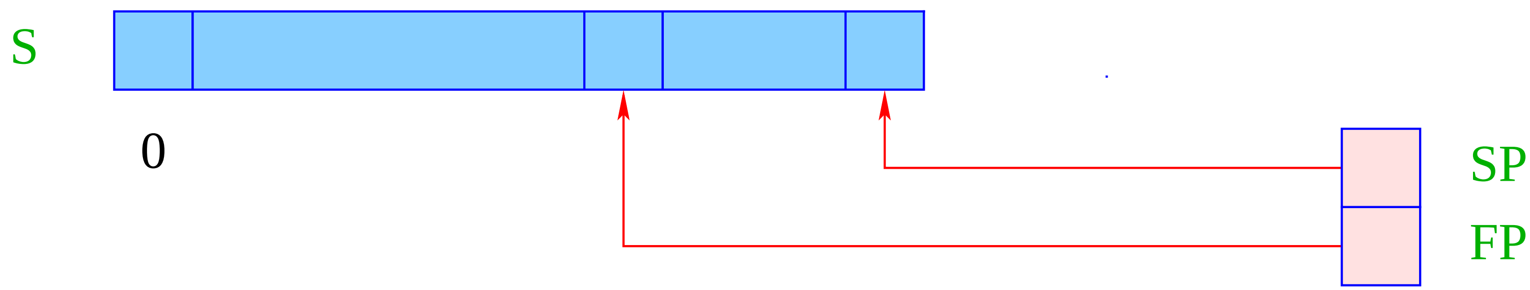
CBN: Arguments are passed unevaluated; they are only evaluated when their value is needed (as in Haskell).

12 Architecture of the MaMa:

We know already the following components:

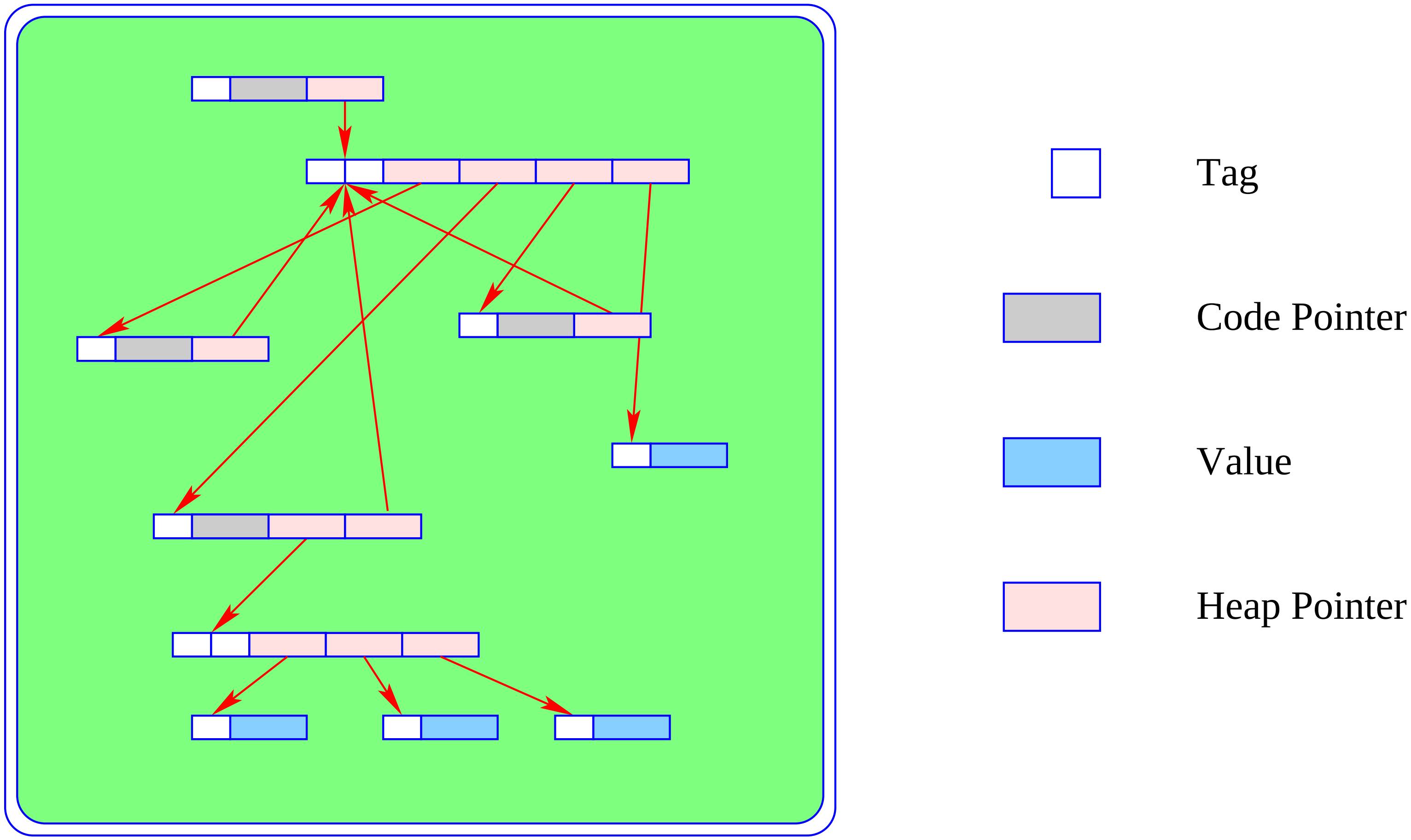


C	=	Code-store – contains the MaMa-program;
		each cell contains one instruction;
PC	=	Program Counter – points to the instruction to be executed next;

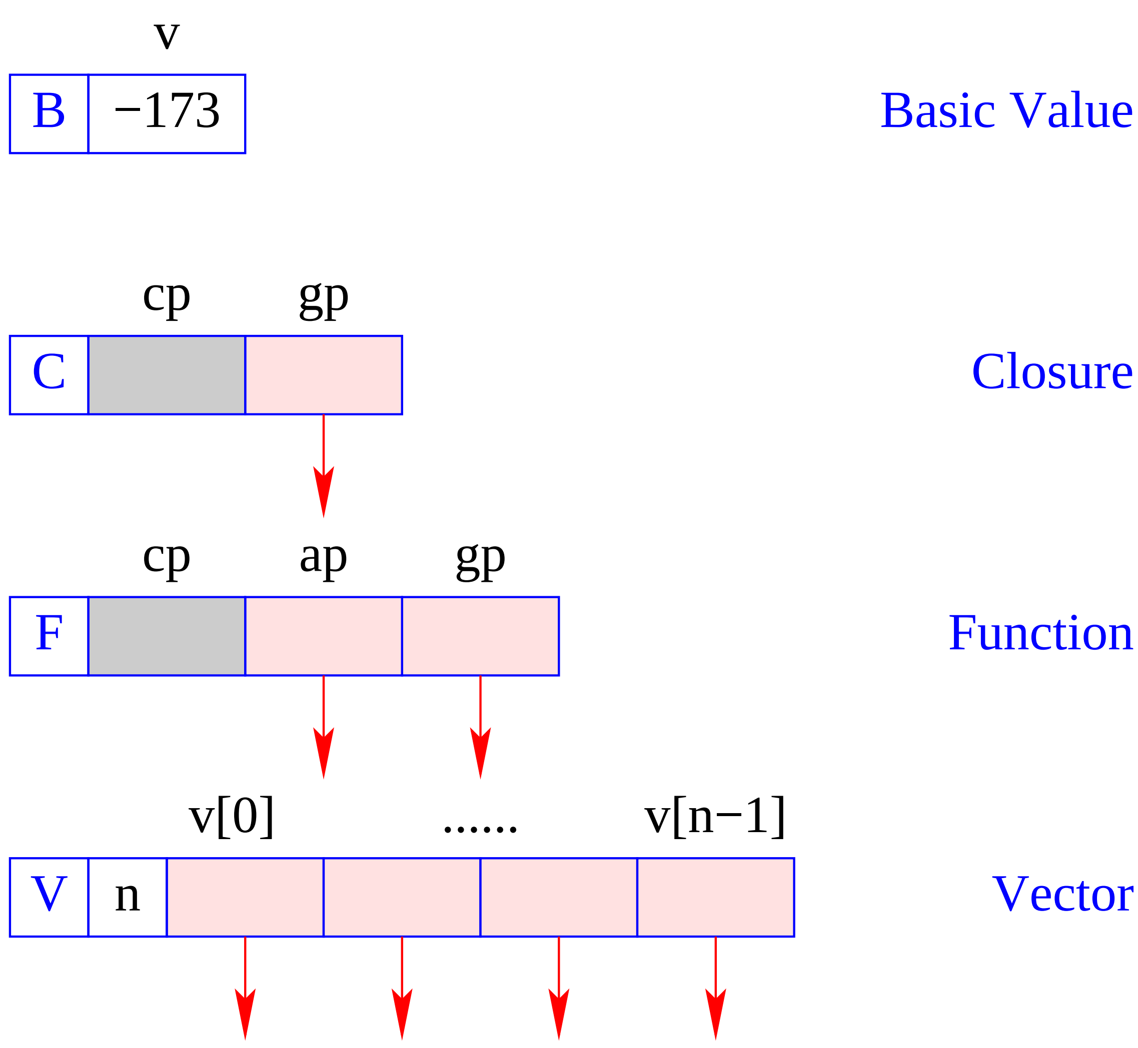


S	=	Runtime-Stack – each cell can hold a basic value or an address;
SP	=	Stack-Pointer – points to the topmost occupied cell;
		as in the CMa implicitely represented;
FP	=	Frame-Pointer – points to the actual stack frame.

We also need a heap H:



... it can be thought of as an abstract data type, being capable of holding data objects of the following form:



The instruction new $(tag, args)$ creates a corresponding object (B, C, F, V) in H and returns a reference to it.

We distinguish three different kinds of code for an expression e:

- $\text{code}_V\ e$ — (generates code that) computes the Value of e, stores it in the heap and returns a reference to it on top of the stack (the normal case);
- $\text{code}_B\ e$ — computes the value of e, and returns it on the top of the stack (only for Basic types);
- $\text{code}_C\ e$ — does not evaluate e, but stores a Closure of e in the heap and returns a reference to the closure on top of the stack.

We start with the code schemata for the first two kinds:

13 Simple expressions

Expressions consisting only of constants, operator applications, and conditionals are translated like expressions in imperative languages:

$$\begin{aligned}
\text{code}_B\; b\; \rho\; \text{sd} &= \text{loadc b} \\
\text{code}_B\; (\Box_1\; e)\; \rho\; \text{sd} &= \text{code}_B\; e\; \rho\; \text{sd} \\
&\quad\; \text{op}_1 \\
\text{code}_B\; (e_1\; \Box_2\; e_2)\; \rho\; \text{sd} &= \text{code}_B\; e_1\; \rho\; \text{sd} \\
&\quad\; \text{code}_B\; e_2\; \rho\; (\text{sd}+1) \\
&\quad\; \text{op}_2
\end{aligned}$$

$$
\begin{array}{lrl}
\text{code}_B\ (\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2)\ \rho\ \text{sd} & = & \text{code}_B\ e_0\ \rho\ \text{sd} \\
 & & \text{jumpz A} \\
 & & \text{code}_B\ e_1\ \rho\ \text{sd} \\
 & & \text{jump B} \\
 & \text{A:} & \text{code}_B\ e_2\ \rho\ \text{sd} \\
 & \text{B:} & \ldots
\end{array}
$$

Note:

- ρ denotes the actual address environment, in which the expression is translated. Address environments have the form:

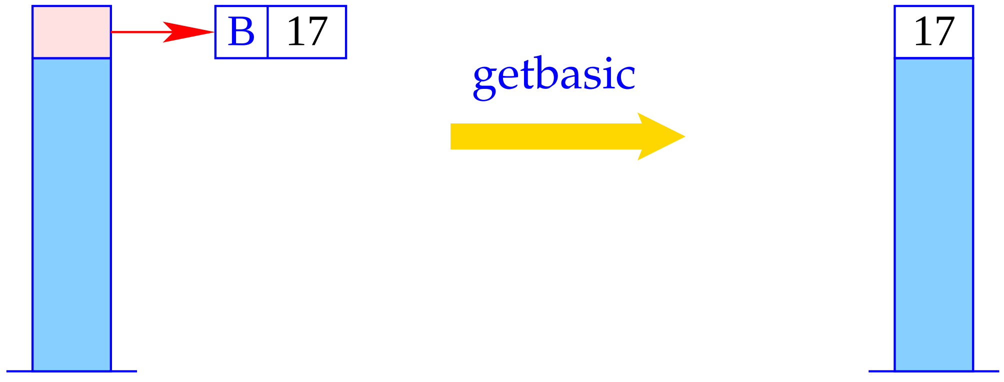
$$\rho : Vars \to \{L, G\} \times \mathbb{Z}$$

- The extra argument sd, the stack difference, *simulates* the movement of the SP when instruction execution modifies the stack. It is needed later to address variables.

- The instructions op_1 and op_2 implement the operators $\Box_1$ and $\Box_2$, in the same way as the the operators neg and add implement negation resp. addition in the CMa.

- For all other expressions, we first compute the value in the heap and then dereference the returned pointer:

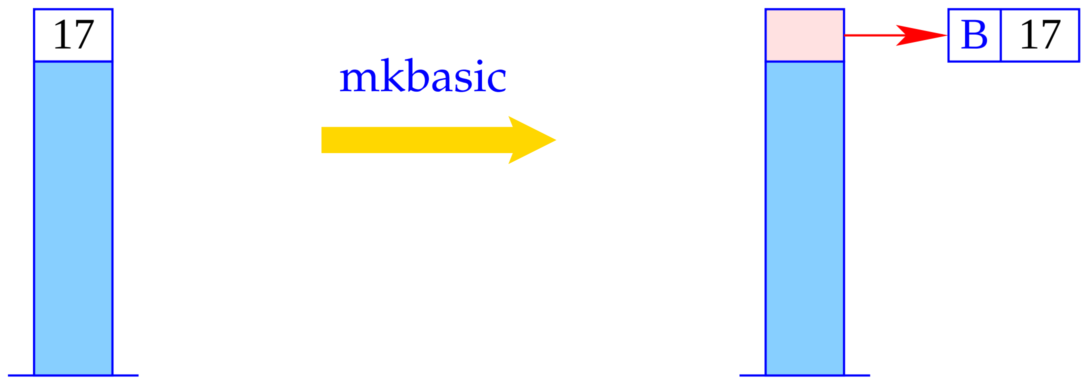
$$\begin{array}{rcl} \text{code}_B\, e\, \rho\, \text{sd} & = & \text{code}_V\, e\, \rho\, \text{sd} \\ & & \text{getbasic} \end{array}$$



if (H[S[SP]] != (B,_))
    Error “not basic!”;
else
    S[SP] = H[S[SP]].v;

For code_V and simple expressions, we define analogously:

$$
\begin{array}{llll}
\text{code}_V\ b\ \rho\ \text{sd} & = & & \text{loadc b; mkbasic} \\
\text{code}_V\ (\square_1\ e)\ \rho\ \text{sd} & = & & \text{code}_B\ e\ \rho\ \text{sd} \\
 & & & \text{op}_1\text{; mkbasic} \\
\text{code}_V\ (e_1\ \square_2\ e_2)\ \rho\ \text{sd} & = & & \text{code}_B\ e_1\ \rho\ \text{sd} \\
 & & & \text{code}_B\ e_2\ \rho\ (\text{sd}+1) \\
 & & & \text{op}_2\text{; mkbasic} \\
\text{code}_V\ (\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2)\ \rho\ \text{sd} & = & & \text{code}_B\ e_0\ \rho\ \text{sd} \\
 & & & \text{jumpz A} \\
 & & & \text{code}_V\ e_1\ \rho\ \text{sd} \\
 & & & \text{jump B} \\
 & & \text{A:} & \text{code}_V\ e_2\ \rho\ \text{sd} \\
 & & \text{B:} & \ldots
\end{array}
$$



S[SP] = new (B,S[SP]);

14 Accessing Variables

We must distinguish between local and global variables.

Example: Regard the function f :

$$\begin{array}{lll} \textbf{let} & c = 5 & \\ & f = \textbf{fn}\ a \quad \Rightarrow & \textbf{let}\ b = a * a \\ & & \textbf{in}\ b + c \\ \textbf{in} & f\ c & \end{array}$$

The function f uses the global variable c and the local variables a (as formal parameter) and b (introduced by the inner **let**).

The binding of a global variable is determined, when the function is constructed (static scoping!), and later only looked up.

Accessing Global Variables

- The bindings of global variables of an expression or a function are kept in a vector in the heap (Global Vector).
- They are addressed consecutively starting with 0.
- When an F-object or a C-object are constructed, the Global Vector for the function or the expression is determined and a reference to it is stored in the gp-component of the object.
- During the evaluation of an expression, the (new) register GP (Global Pointer) points to the actual Global Vector.
- In constrast, local variables should be administered on the stack ...

$\Longrightarrow$ General form of the address environment:

$$\rho : \mathit{Vars} \to \{L, G\} \times \mathbb{Z}$$

Accessing Local Variables

Local variables are administered on the stack, in stack frames.

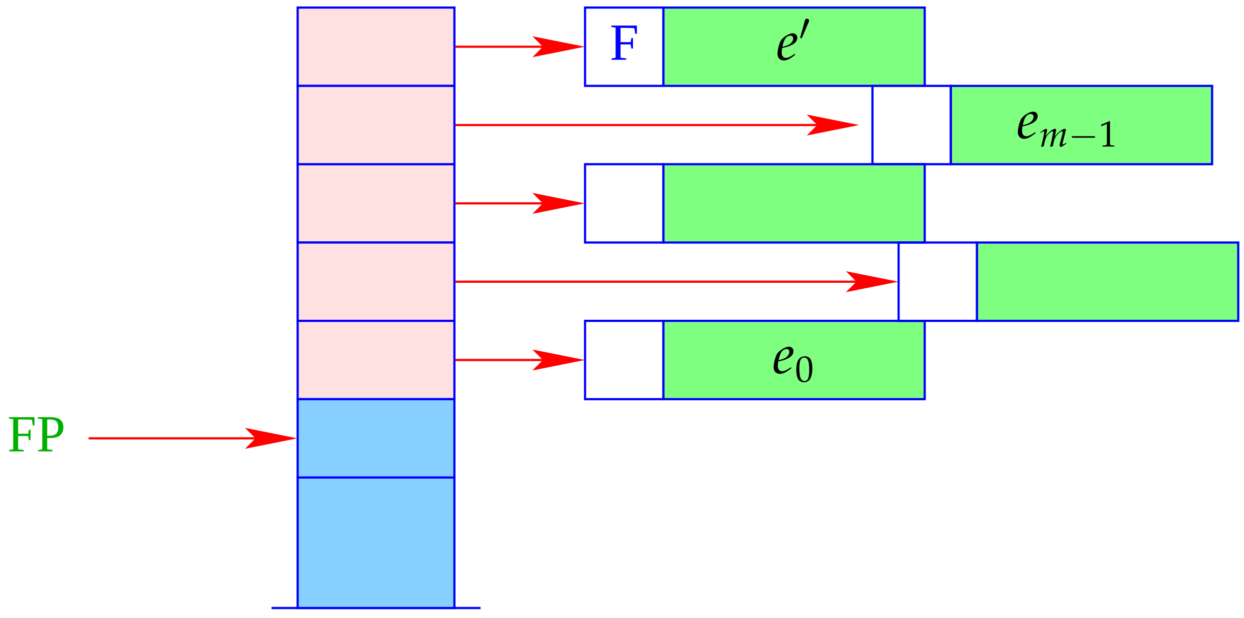
Let $e \equiv e' \; e_0 \; \ldots \; e_{m-1}$ be the application of a function e' to arguments $e_0, \ldots, e_{m-1}$.

Warning:

The arity of e' does not need to be m :-)

- PuF functions have curried types, $f : t_1 \to t_2 \to \ldots \to t_n \to t$
- f may therefore receive less than n arguments (under supply);
- f may also receive more than n arguments, if t is a functional type (over supply).

Possible stack organisations:

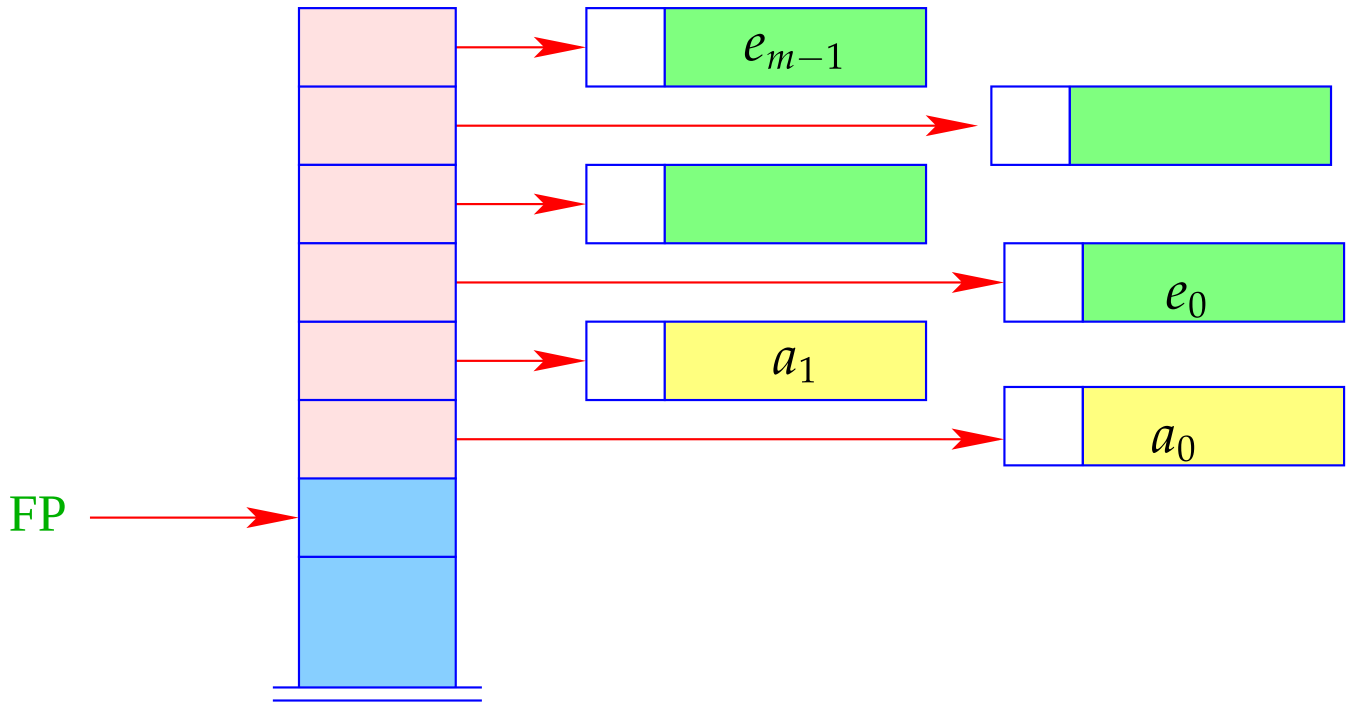


+ Addressing of the arguments can be done relative to FP

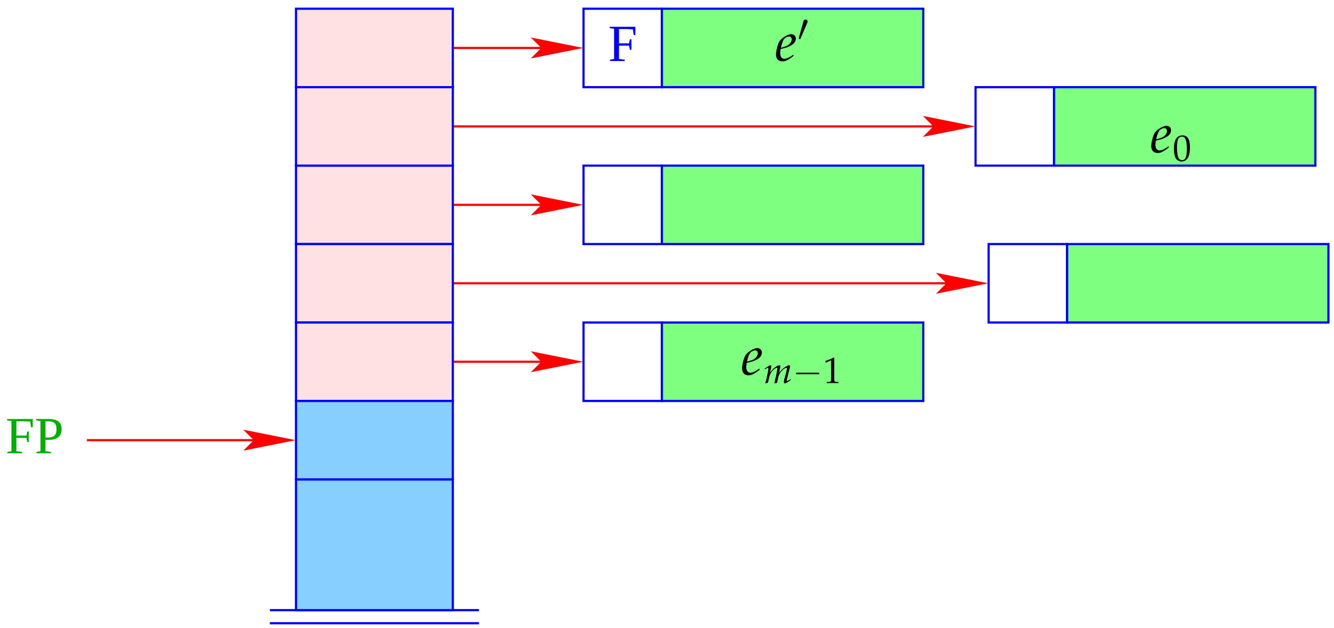
– The local variables of e' cannot be addressed relative to FP.

– If e' is an n-ary function with $n < m$, i.e., we have an over-supplied function application, the remaining $m - n$ arguments will have to be shifted.

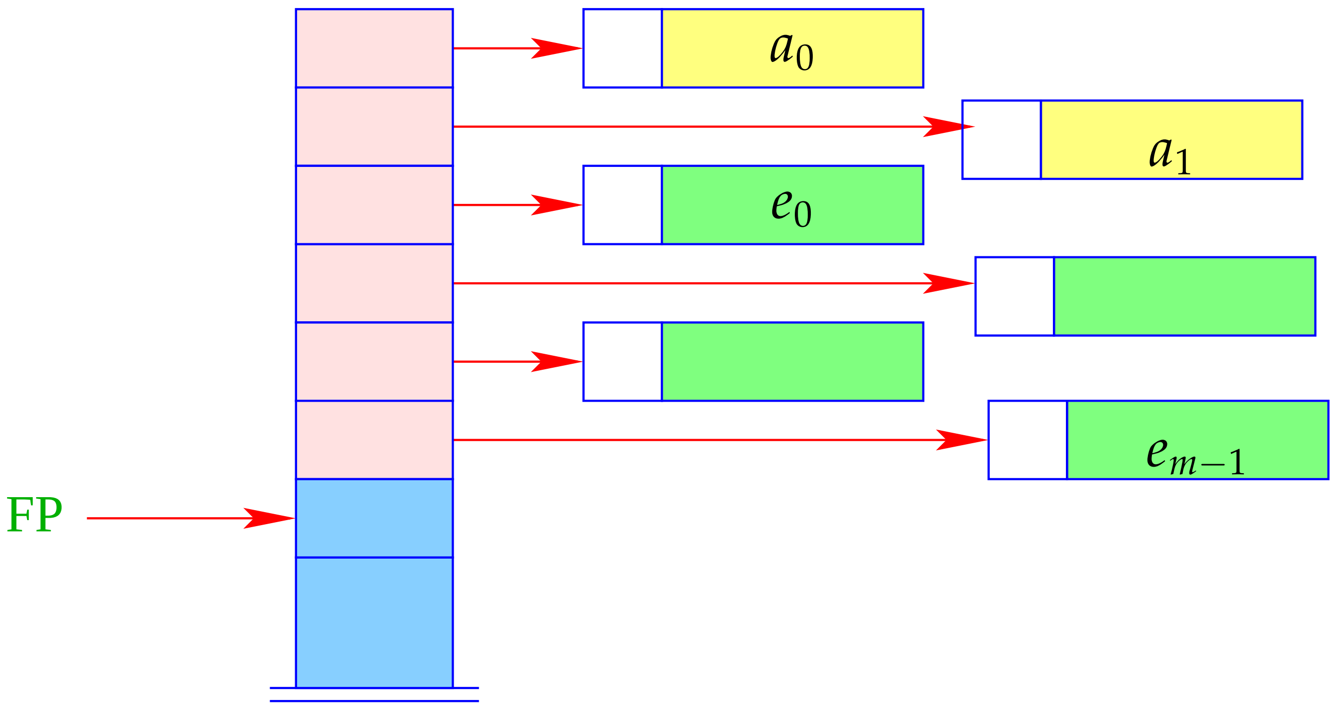
– If e' evaluates to a function, which has already been partially applied to the parameters $a_0, \ldots, a_{k-1}$, these have to be sneaked in underneath e_0:



Alternative:



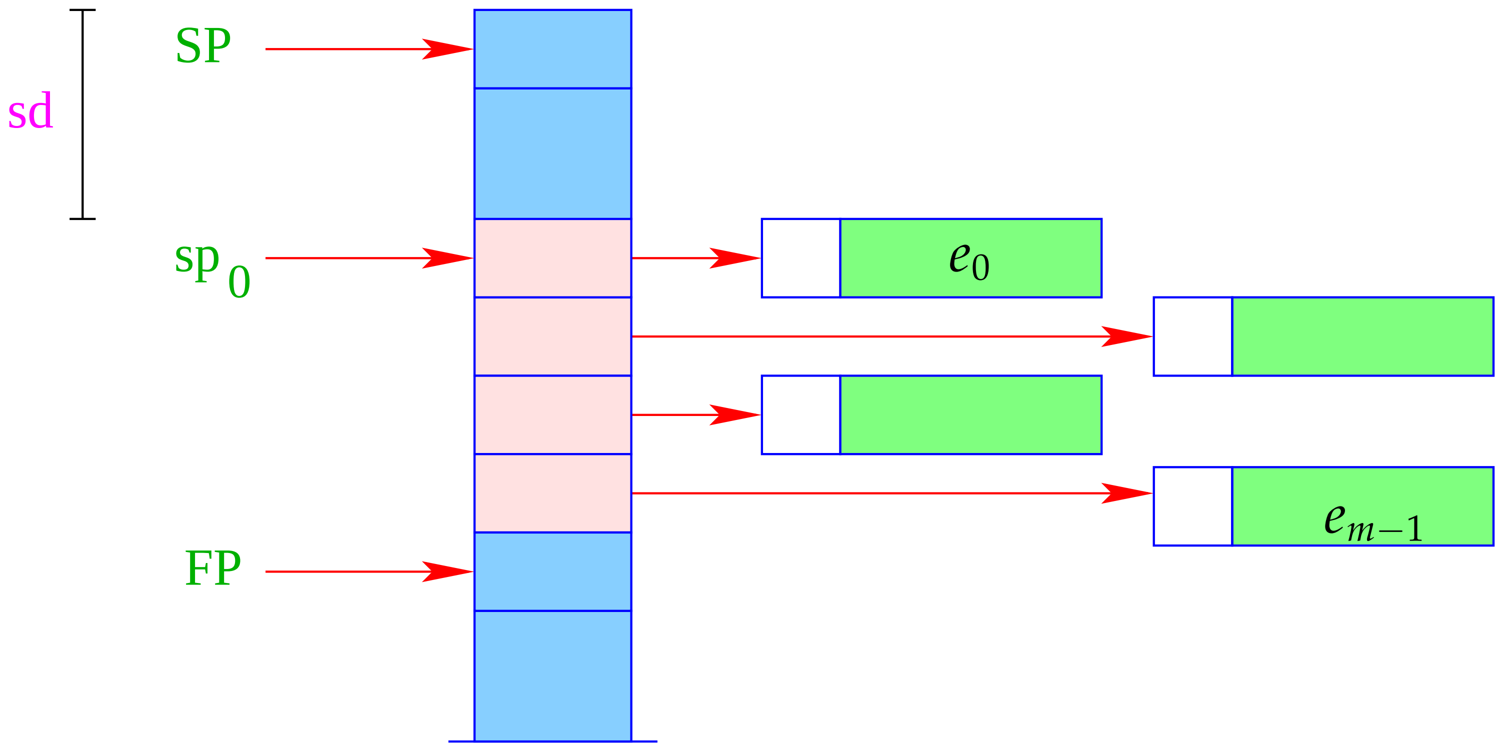
+ The further arguments $a_0, \ldots, a_{k-1}$ and the local variables can be allocated above the arguments.



– Addressing of arguments and local variables relative to FP is no more possible. (Remember: m is unknown when the function definition is translated.)

Way out:

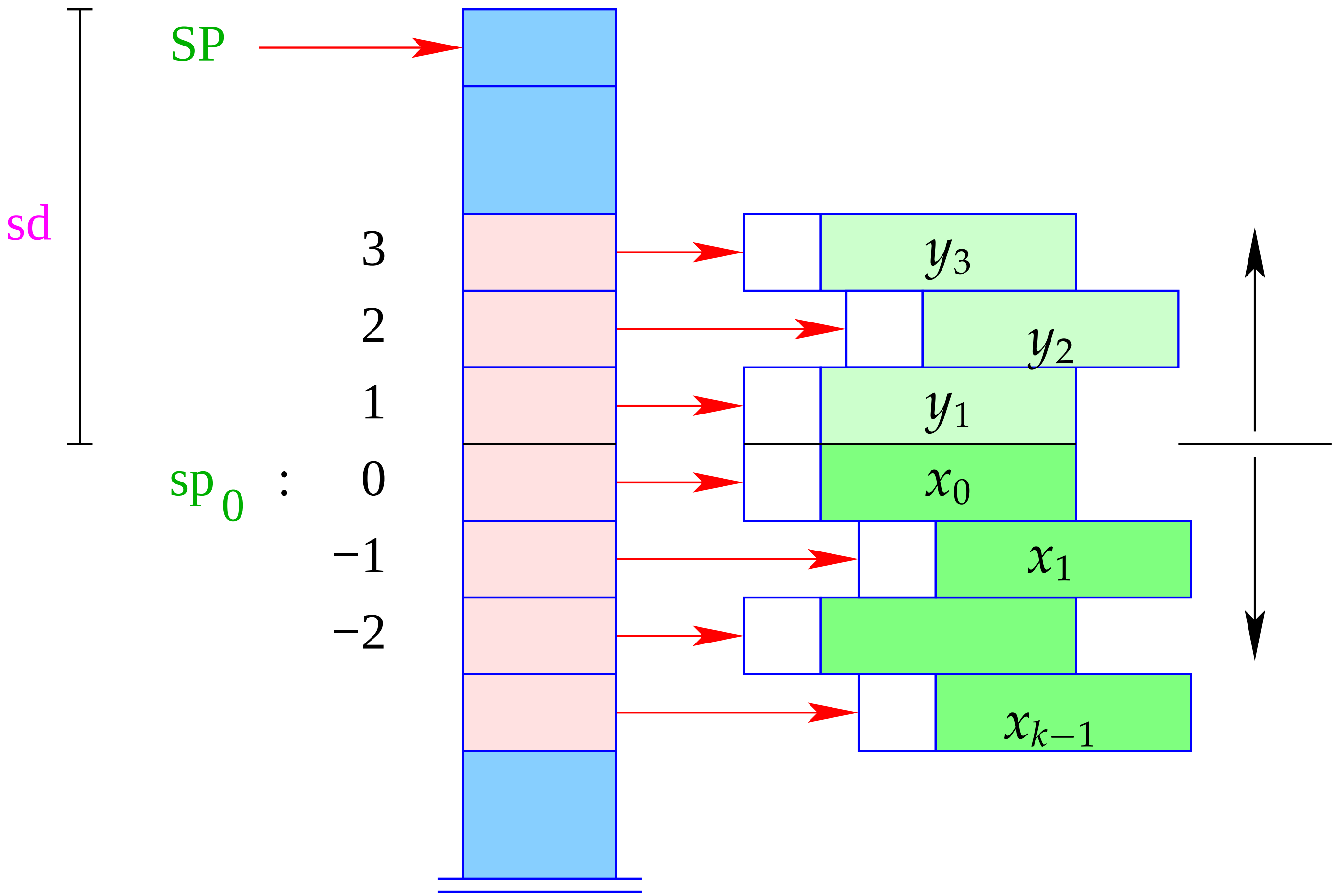
- We address both, arguments and local variables, relative to the stack pointer SP !!!
- However, the stack pointer changes during program execution...



- The differerence between the current value of SP and its value sp_0 at the entry of the function body is called the stack distance, sd.
- Fortunately, this stack distance can be determined at compile time for each program point, by simulating the movement of the SP.
- The formal parameters $x_0, x_1, x_2, \ldots$ successively receive the non-positive relative addresses $0, -1, -2, \ldots$, i.e., $\rho\, x_i = (L, -i)$.
- The absolute address of the i-th formal parameter consequently is

$$\mathrm{sp}_0 - i = (\mathrm{SP} - \mathrm{sd}) - i$$

- The local **let**-variables $y_1, y_2, y_3, \ldots$ will be successively pushed onto the stack:



- The y_i have positive relative addresses $1, 2, 3, \ldots$, that is: $\rho\ y_i = (L, i)$.
- The absolute address of y_i is then $\text{sp}_0 + i = (\text{SP} - \text{sd}) + i$

With CBN, we generate for the access to a variable:

$$\text{code}_V\; x\; \rho\; \text{sd} \quad = \quad \begin{array}{l} \text{getvar}\; x\; \rho\; \text{sd} \\ \text{eval} \end{array}$$

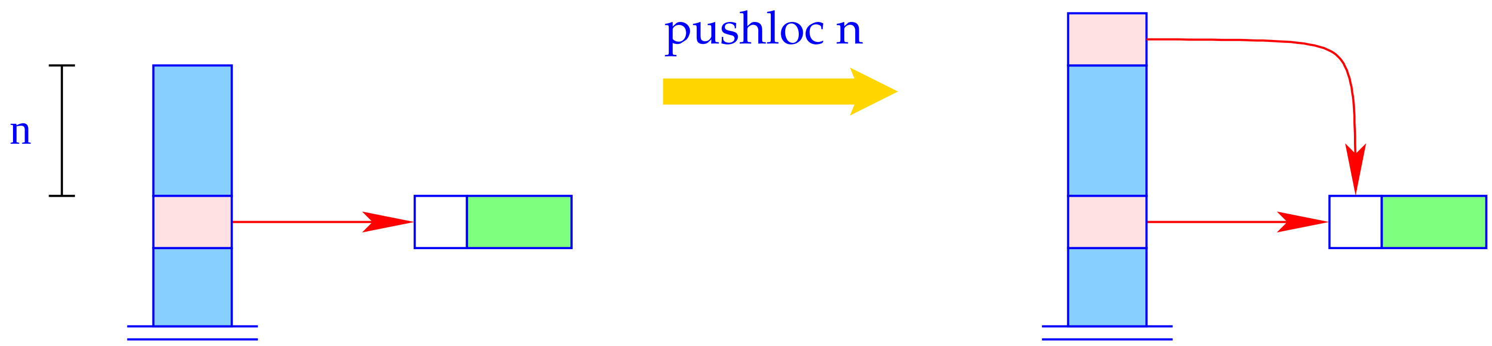
The instruction eval checks, whether the value has already been computed or whether its evaluation has to yet to be done ($\Longrightarrow$ will be treated later :-)

With CBV, we can just delete eval from the above code schema.

The (compile-time) macro getvar is defined by:

$$\begin{array}{lll} \text{getvar}\; x\; \rho\; \text{sd} & = & \textbf{let}\; (t, i) = \rho\; x\; \textbf{in} \\ & & \textbf{case}\; t\; \textbf{of} \\ & & \quad L \Rightarrow \text{pushloc}\; (\text{sd} - i) \\ & & \quad G \Rightarrow \text{pushglob i} \\ & & \textbf{end} \end{array}$$

The access to local variables:



S[SP+1] =S[SP - n]; SP++;

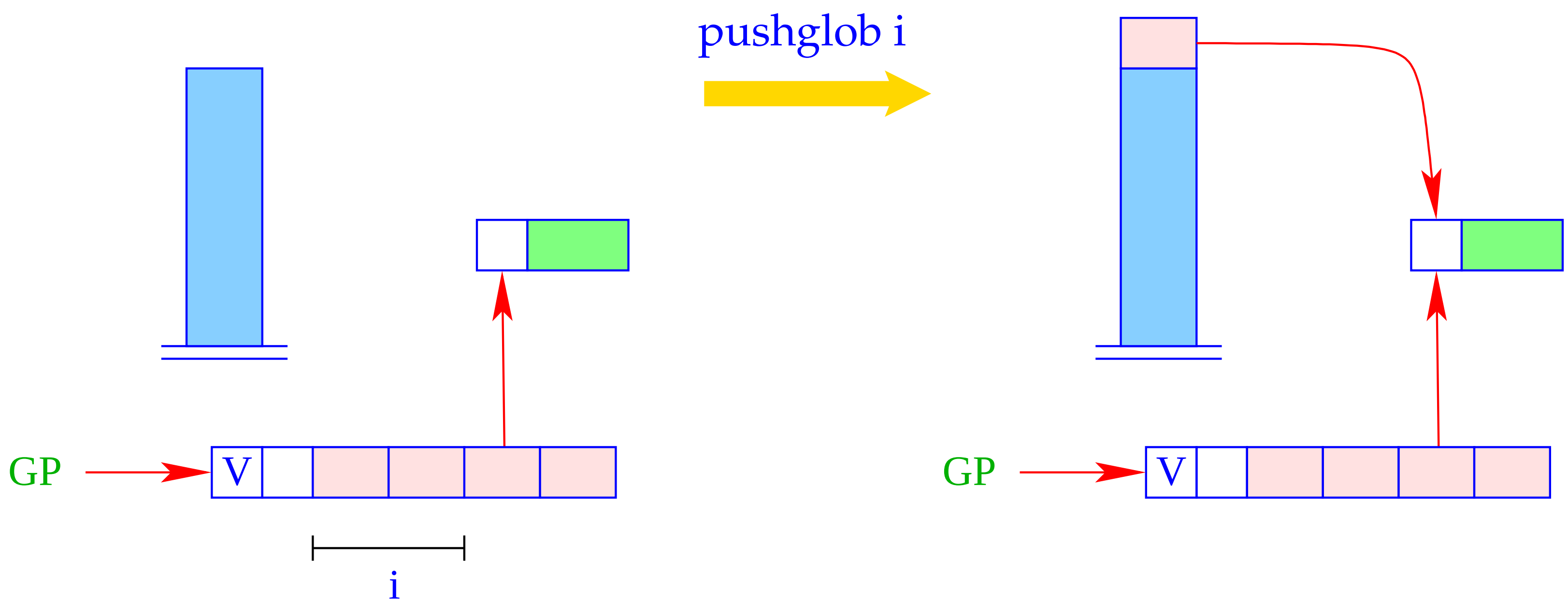
Correctness argument:

Let sp and sd be the values of the stack pointer resp. stack distance before the execution of the instruction. The value of the local variable with address i is loaded from $S[a]$ with

$$a = \mathrm{sp} - (\mathrm{sd} - i) = (\mathrm{sp} - \mathrm{sd}) + i = \mathrm{sp}_0 + i$$

... exactly as it should be :-)

The access to global variables is much simpler:



SP = SP + 1;
S[SP] = GP→v[i];

Example:

Regard $e \equiv (b + c)$ for $\rho = \{b \mapsto (L, 1), c \mapsto (G, 0)\}$ and sd $= 1$.

With CBN, we obtain:

$$
\begin{array}{lllll}
\text{code}_V\; e\; \rho\; 1 & = & \text{getvar}\; b\; \rho\; 1 & = & 1 \quad \text{pushloc } 0 \\
 & & \text{eval} & & 2 \quad \text{eval} \\
 & & \text{getbasic} & & 2 \quad \text{getbasic} \\
 & & \text{getvar}\; c\; \rho\; 2 & & 2 \quad \text{pushglob } 0 \\
 & & \text{eval} & & 3 \quad \text{eval} \\
 & & \text{getbasic} & & 3 \quad \text{getbasic} \\
 & & \text{add} & & 3 \quad \text{add} \\
 & & \text{mkbasic} & & 2 \quad \text{mkbasic}
\end{array}
$$

15 let-Expressions

As a warm-up let us first consider the treatment of local variables :-)

Let $e \equiv \textbf{let } y_1 = e_1; \ldots; y_n = e_n \textbf{ in } e_0$ be a **let**-expression.

The translation of e must deliver an instruction sequence that

- allocates local variables $y_1, \ldots, y_n$;
- in the case of
  CBV: evaluates $e_1, \ldots, e_n$ and binds the y_i to their values;
  CBN: constructs closures for the $e_1, \ldots, e_n$ and binds the y_i to them;
- evaluates the expression e_0 and returns its value.

Here, we consider the non-recursive case only, i.e. where y_j only depends on $y_1, \ldots, y_{j-1}$. We obtain for CBN:

$$
\begin{array}{rcl}
\text{code}_V\; e\; \rho\; \text{sd} & = & \text{code}_C\; e_1\; \rho\; \text{sd} \\
& & \text{code}_C\; e_2\; \rho_1\; (\text{sd}+1) \\
& & \ldots \\
& & \text{code}_C\; e_n\; \rho_{n-1}\; (\text{sd}+n-1) \\
& & \text{code}_V\; e_0\; \rho_n\; (\text{sd}+n) \\
& & \text{slide n} \qquad\qquad // \text{ deallocates local variables}
\end{array}
$$

where $\rho_j = \rho \oplus \{y_i \mapsto (L, \text{sd}+i) \mid i = 1, \ldots, j\}$.

In the case of CBV, we use code_V for the expressions $e_1, \ldots, e_n$.

Warning!

All the e_i must be associated with the same binding for the global variables!

Example:

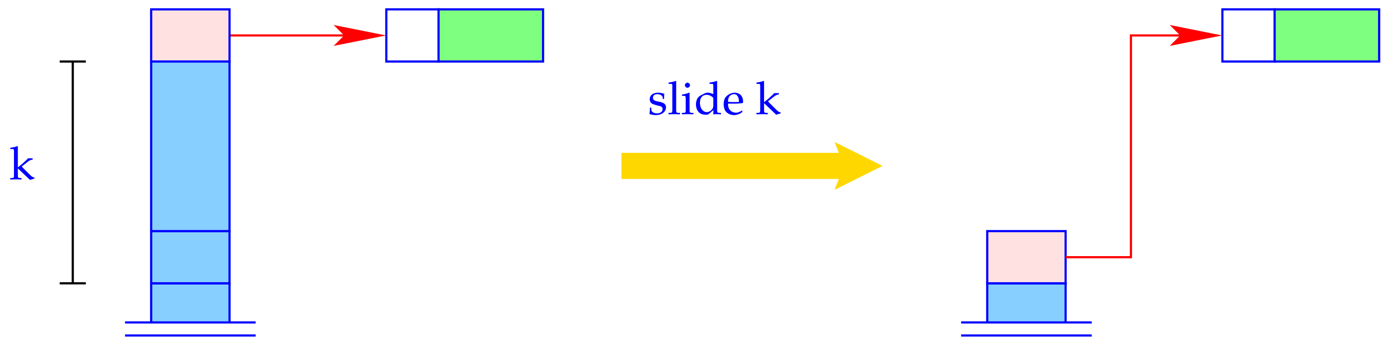
Consider the expression

$$e \equiv \textbf{let } a = 19; b = a * a \textbf{ in } a + b$$

for $\rho = \emptyset$ and sd $= 0$. We obtain (for CBV):

0	loadc 19	3	getbasic	3	pushloc 1
1	mkbasic	3	mul	4	getbasic
1	pushloc 0	2	mkbasic	4	add
2	getbasic	2	pushloc 1	3	mkbasic
2	pushloc 1	3	getbasic	3	slide 2

The instruction slide k deallocates again the space for the locals:



S[SP-k] = S[SP];
SP = SP - k;